Semantics and Security Issues in JavaScript

Stéphane Ducasse, Nicolas Petton, Guillermo Polito, Damien Cassou

Semantics and Security Issues in JavaScript

Sémantique et problèmes de sécurité en JavaScript

Deliverable Resilience FUI 12:
7.3.2.1 Failles de sécurité en JavaScript
/ JavaScript security issues

Stéphane Ducasse, Nicolas Petton, Guillermo Polito, Damien Cassou

Version 1.2 - December 2012

Deliverable: 7.3.2.1 Failles de sécurité en Javascript

Title: Semantics and Security Issues in JavaScript

Titre: Sémantique et problèmes de sécurité en JavaScript

Version: 1.2

Authors: Stéphane Ducasse, Nicolas Petton, Guillermo Polito, Damien Cassou

Description de la tâche 7.3.2.1 / Task 7.3.2.1 Description:

Les environnements d'exécution Javascript comportent des failles de sécurité liées à la sémantique d'ECMAScript ainsi qu'à la présence de certains constructeurs. D'autres failles sont liées à la mise en oeuvre pratique d'ECMAScript par les navigateurs. L'objectif de cette tâche est d'identifier et classer les types de failles de sécurité dans le monde Javascript. On consultera notamment les travaux de Miller autour de Caja et ainsi que les approches proposées par Seaside un framework pour le développement d'applications sécurisées en Smalltalk. L'objectif de la tâche est de formaliser les mécanismes de sécurité en Javascript de façon à augmenter la connaissance des partenaires et des utilisateurs de Javascript sur ce sujet.

Contents

Abstract

There is a plethora of research articles describing the deep semantics of JavaScript. Nevertheless, such articles are often difficult to grasp for readers not familiar with formal semantics. In this report, we propose a digest of the semantics of JavaScript centered around security concerns. This document proposes an overview of the JavaScript language and the misleading semantic points in its design. The first part of the document describes the main characteristics of the language itself. The second part presents how those characteristics can lead to problems. It finishes by showing some coding patterns to avoid certain traps and presents some ECMAScript 5 new features.

Chapter1. Web architecture basics

In this chapter we present some basics principles of common web architecture and in which JavaScript applications are deployed.

1.1 Client-Server basics

The web is based on the client-server architectural pattern:

- the client role is played by a web browser with limited resources and technologies – often HTML, CSS and JavaScript. Its usual main responsibilities are user interface and interaction as well as data validation.
- the server role is fulfilled by a program implemented in a wide variety of technologies, with a controlled set of resources. Its usual responsibilities are to serve pages to web-browsers, enforce business rules, and persist and validate data.

The client and server processes communicate though the HTTP protocol [Mog02]: the client makes HTTP requests and, for each, the server answers with an HTTP response, as illustrated in Figure 1.1. When the response arrives, the web browser normally discards the current content and loads the new one, causing a full *refresh* of the user interface. Since HTTP is a stateless protocol – requests are independent from each others – several mechanisms were built to maintain state between a client and a server (*e.g.,* cookies, see section 1.3).

As computers running *web browsers* become more and more powerful, there is a tendency to move responsibilities from the server to the client (*e.g.,* page creation and business rules enforcement). These new responsibilites are mostly implemented in JavaScript and JavaScript-based languages.

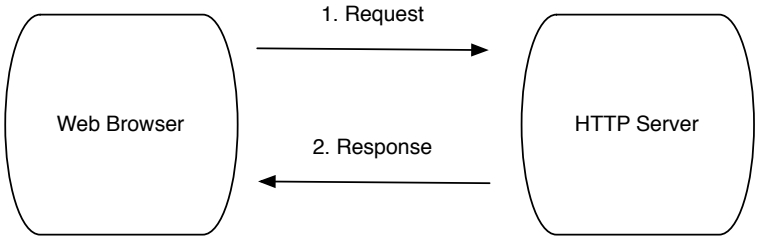

Figure 1.1: An HTTP request and response between a web browser and a web server

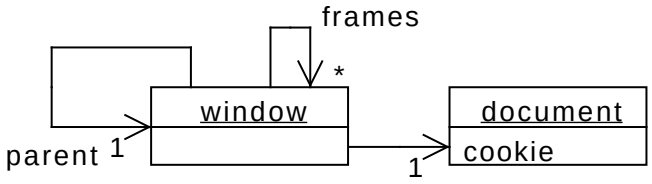

Figure 1.2: The JavaScript model of a web page

1.2 Web Browser window insfrastructure

A web page is composed of several html widgets including frames and iframes, which insert an existing web page into the current one. *Frames* and *iFrames* are considered web pages by themselves, therefore having the ability to contain other ones, following the composite design pattern [GHJV95].

While loading a web page, a web browser generates an internal structure of JavaScript objects. Each web page is represented as a window object referencing its parent and children through the parent and frames properties.

Each web page also references a document object, which is the root of the *DOM* – Document Object Model – representation of the web page tags.

Web browser JavaScript implementation differences

As each web browser has its own implementation of the *DOM* and JavaScript, one of the hardest tasks of JavaScript development is to build browser-agnostic applications. There exist several libraries to reduce the difficulty of this task such as *extJS*[1] ,*jQuery*[2] and *scriptaculous*.[3]

1.3 Cookies

A cookie is a string that is exchanged between the client and the server [MFF01]. Cookies are normally stored on the client machines to let web application state survive after the client process is stopped (*i.e.,* the browser or the browsing tab is closed).

Cookies are key-value pairs associated with the equals character '=' and separated from each other with a semi colon (*e.g.,*, *"name=john;expires=2013-01-*

[1] http://www.sencha.com/products/extjs/
[2] http://jquery.com/
[3] http://script.aculo.us/

15T21:47:38Z"). The expiration date tells the browser when to delete it. A cookie can contain other properties like domain and path, that tells the browser in which requests it should be exchanged, which by default are the current domain and path.

Cookies can be changed from JavaScript code accessing the cookie property of the current document object. Cookies can contain sensible data (*e.g.,* emails and passwords) and must be studied for a security analysis.

1.4 AJAX

AJAX stands for Asynchronous JavaScript and XML: it is a mechanism to send XMLHttpRequest requests to a web server. XMLHttpRequest is used to retrieve data from a URL without reloading the complete web page [MvD07]. The request is sent by the client through a JavaScript API and handled asynchronously. Formerly, AJAX was used to exchange information in XML format, but XMLHttpRequest can be used to transfer any kind of data. Nowadays, one of the most used formats is *JSON*,[4] which stands for JavaScript Object Notation.

Illustrated in Example 1.1, the "test.json" service is requested to the server via AJAX, and a function is used to determine how the response will be handled at the time it arrives.

```
$.ajax({
    url: "http://www.webcompany.com/api/test.json",
}).done(
    // Function is evaluated after the json service responds:
function() {
    // adds a 'class' attribute to current HTML node
    $(this).addClass("done");
});
```

Example 1.1: AJAX request performed with JQuery Library

AJAX is becoming more and more popular as the client machines become more powerful and can perform heavier processing tasks.

[4] http://www.json.org

Chapter2. JavaScript object model in a nutshell

In this chapter we present key aspects of ECMAScript 3 and 5[1] to provide a common context. This document has been written in the context of the Resilience project. This study is focused on the semantic aspects of JavaScript in the context of securing untrusted third party JavaScript code in a web browser. In this context, one of the key aspects is to forbid the access to the window object (the global object of the web browser frame, see 3.2). Accessing the window object of the frame would lead to possible page-data leak (window is the root of the DOM of the page), sensitive information leak (from the window object private information in cookies can be accessed), and even user actions sniffing (having access to DOM elements means that one can add listeners to DOM events[2]). In the next deliverable we will describe the state of the art of JavaScript sandboxing. One of the different aspects in JavaScript sandboxing is to securely forbid access to potentially sensitive objects, including window, the root object.

2.1 Objects: the basic building block of JavaScript

JavaScript is a loosely-typed dynamic programming language where everything is an object. Each object contains a set of properties (also named slots) that represent data (other objects) and operations (function objects). These properties are always public and can be accessed and modified using the dot or squared-bracket notation:

```
// We create a new empty object
var person = new Object();

// Write a property
person.age = 5;
person['age'] = 5; // equivalent

// Read a property and store it into some variable
var theAge = person.age;
var theAge = person['age']; // equivalent
```

Example 2.1: Property access

In this script we create a new object that we assign to the variable person. Then the expression person.age = 5 adds a property to the newly created object and stores the value 5 into it. Note that objects are hash tables. The expression {a:

[1] ECMAScript is the standard JavaScript is based on.

[2] Adding event listeners to DOM objects allows an attacker to catch user interactions and inputs.

0} thus creates an object with one property named a with value 0. Because of this, properties can be accessed and set using both the person['age'] and person.age notations.

2.2 Property access

ECMAScript's property lookup is done at runtime (see Example 2.2) and never throws errors. Example 2.2 defines an object object containing two properties a and b with values 10 and 5 respectively. The function get is then defined. This function takes as argument a property's name and returns the corresponding property's value of the object. In the last line of Example 2.2, the function get is called twice to access the values of the properties a and b.

```
var object = {a : 10, b: 5};
var get = function(property) {
  return object[property]
};
get("a") + get("b"); // answers 15
```

Example 2.2: Property lookup at runtime

If a property does not exist, the undefined object is returned, as shown in Example 2.3.

```
var object = {a: 1, b: 2};
object.c // answers 'undefined'
```

Example 2.3: Property access for nonexistent properties

Example 2.4 shows how to update, create and delete properties. The first instruction sets 10 to the property a of object {a: 0}. Therefore the expression returns an object with the property a with 10 as a value.

The second instruction shows that if a property does not exist then it will be created automatically. Hence, {a: 0}[b]=10 returns an object with two properties a and b.

Finally, the third instruction removes the property b of the object using the delete keyword.

```
// Updating a property
{a: 0}[a] = 10; // answers {a: 10}

// Creating a new property using the same syntax
{a: 0}[b] = 10; // answers {a: 0, b: 10}
```

```
// Deleting a property
delete {a: 0, b: 5}[b]; // answers {a: 0}
```

Example 2.4: Updating, creating and deleting properties

2.3 Functions: behavioral units

Functions are the behavioral units of JavaScript. A function performs a side effect (*e.g.,* alters an object) and/or returns a value.

Functions can be defined via a function declaration or a function expression. A function declaration defines a function at compile time, as seen in Example 2.5. A function expression creates a new function at runtime. Function expressions are also called anonymous functions, since they are not associated with any name. The creation of a function from a function expression is illustrated in Example 2.6.

```
function sum(a, b){
   return a + b;
}
```

Example 2.5: A function declaration has a name

```
var sum = function (a, b){
   return a + b;
}
```

Example 2.6: An anonymous function that is assigned to a variable

A function can be called by passing arguments between parenthesis as shown in Example 2.7.

```
sum(1, 2);
someFunctionWithNoArguments();
```

Example 2.7: Calling a function

Functions are important because they constitute the basic building block of code execution in a JavaScript program.

2.4 Programmatic function evaluation

The built-in call() and apply() functions are two methods of the Function object.[3] These functions are thus called on other functions and execute them. These

[3]In JavaScript, the words 'method' and 'function' are equivalent. The former is particularly used when the function is owned by an object as one of its properties.

functions both take the receiver of the function to execute as first parameter and arguments of the function to execute as the other parameters. When using these methods, the pseudo-variable this in the function to execute is bound to the first parameter of call() and apply(). call() receives the arguments of the function to execute as comma-separated values whereas apply() receives an array (see Example 2.8).

```
function someGlobalFunction(value1, value2){
  this.value1 = value1;
  this.value2 = value2;
}

// The regular invocation binds 'this' to the global object as
// we will see later on

someGlobalFunction(5,6);
window.value1 // answers 5

var someObject = new Object();
someGlobalFunction.call(someObject, 5, 6);
someObject.value1 // answers 5

someGlobalFunction.apply(someObject, [5, 6]); // equivalent
```

Example 2.8: The apply() and call() methods

Turning text into function at runtime. Using the eval() built-in function, it is possible to evaluate a string at runtime. eval() makes static analysis of programs difficult [RHBV11]: at runtime a string can be composed and evaluated without restriction in the global environment and scope.

```
var a = 2;
eval('a++');
a // answers 3
```

Example 2.9: Evaluating code from a string

2.5 Object methods

In JavaScript, methods are just functions attached to an object as can be seen in Example 2.10.

```
var person = new Object();
person.name = 'John';
person.surname = 'Foo';
```

```
person.getFullName = function (){
    return this.name + ' ' + this.surname;
}
```

Example 2.10: Adding methods to an object

In a method, a developer can refer to the owner object using the this keyword. We will see later that this has a different semantics than in Java or Smalltalk since depending on how the function is accessed this can be bound to the object owner of the property or any other object (see section 3.3).

2.6 Higher-order functions

Functions in JavaScript, as first class citizens, can be passed as parameters to functions and returned from functions. *Higher-order functions* are most commonly seen when implementing filtering, selection and sorting algorithms to make them independent of their content. In Example 2.11, we define a new property inherited by all arrays that is a function to filter elements of the receiver array according to a given criteria. Then, we define a function isEven that will play the role of the criteria and an array of numbers. The last statement calls the new filter property function on array with the isEven criteria.

```
Array.prototype.filter = function (criteria){
    newArray = new Array();
    for (var i = 0; i < this.length; i++) {
        // we keep the elements of the array that respects
        // a certain criteria
        if (criteria(this[i]))
            newArray.push(this[i]);
    }
    return newArray;
}

var isEven = function(elem) { return (elem % 2) == 0; };
var array = new Array(9, 58, 42, 12, 1001, 1000);
array.filter(isEven); // answers [58, 42, 12, 1000]
```

Example 2.11: Extending Arrays with a higher-order filter function

2.7 Object constructors

Constructors are used to structure object creation in ECMAScript 3[MMT08, GSK10]. A constructor is a standard function object whose name is by convention capitalized to indicate to the programmer that the function must not be directly called. The new keyword is used to invoke a function as a constructor. Using the

new keyword instantiates an object and executes the constructor on that object, binding the pseudo-variable this to the newly created object.

```
var Animal = function (name) {
   this.name = name;
   this.describe = function() {
      return this.name + ', an animal';
   }
};

//Invoking the constructor with the 'new' keyword
var animal = new Animal("pilou");

animal.name; // 'pilou'
animal.describe() // 'pilou, an animal'
```

Example 2.12: The new keyword

2.8 Object core properties

There are three properties that are key to the execution of any JavaScript application:

- constructor is a property that contains the reference to the function that was used to create the object using the new keyword. In Figure 2.1, the animal object has a property constructor containing a reference to the Animal constructor that was used to create animal.
- prototype is a property that is used by constructors to initialize the inheritance chain (modeled by the __proto__ property) of objects they create. This property only makes sense on function objects that are used as constructors. In Figure 2.1, the Animal constructor has a property prototype containing a reference to an object (with an isAnAnimal property) that will be the parent of all objects Animal creates.
- __proto__ is a property that contains a reference to an object (the parent) where properties will be looked up when not present in the receiver. When an object is created, the value of its __proto__ property is initialized to the prototype property of the constructor function used to create this object. In Figure 2.1, the animal object has as parent the object whose only property is isAnAnimal: this object is the prototype of the Animal constructor. *Attention:* this property is not always visible from a developer's point of view depending on the JavaScript implementation: you should avoid manipulating __proto__ if you want to write portable code.

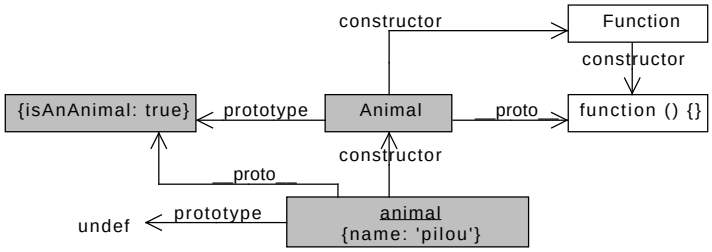

Figure 2.1: The 3 core properties of objects (constructor, prototype, __proto__). Boxes in gray represent objects created by the code of Example 2.13 whereas boxes in white represent objects provided by JavaScript.

Example 2.13 demonstrates that the __proto__ property is initialized to the constructor's prototype property value. First, a function is defined and assigned to a new variable Animal. The prototype of new functions is always an empty object. In this case, we add the property isAnAnimal to this empty object. All objects created from the Animal function will then inherit the isAnAnimal property. The variable animal is then set to a new Animal. The rest of the statements describe the relationship between animal and its constructor Animal.

```
var Animal = function (name) { this.name = name; };
Animal.prototype.isAnAnimal = true;

var animal = new Animal("pilou");

animal.constructor == Animal; // answers true
animal.__proto__ == animal.constructor.prototype; // answers true
animal.isAnAnimal; // answers true
Animal.isAnAnimal; // answers false, 'isAnAnimal' is not a property of
                   // the constructor but of objects it constructs
```

Example 2.13: Defining a constructor so that Figure 2.1 can show the core properties involved

Chapter3. Key subtle JavaScript points

In the following we present the subtle aspects of JavaScript that have an impact on security such as unbound variables [MMT08, Dav06].

3.1 Variables and scope

All objects inside a web browser frame share the same environment without any restriction. This is a security problem because it allows dynamic modifications of the program and full access to all the objects defined inside this environment.

To declare a variable local in a particular scope such as a function scope the var keyword is needed, as illustrated in Example 3.1. Here the variable myLocalVariable is local and only accessible from within the body of the function myFunction.

```
function myFunction(arg) {
  var myLocalVariable = arg + 5;
  return myLocalVariable;
}
```

Example 3.1: declaring a variable local to the function

Not using var causes our program to create a global variable (see Example 3.2) [MMT08, Dav06, GSK10].

```
(function () { globalVar = 'setting global'; })()

window.globalVar // answers 'setting global'
```

Example 3.2: using a global variable

In this example, globalVar becomes a property of the global environment window.

3.2 The window object

The window object represents the window of the current HTML document. Each web browser's tab, frame, or window has its own window object.

The window object is the global object of the JavaScript environment. Each function that is not attached to another object is attached to window, as illustrated in Example 3.3. This behavior combined with the dynamic binding of the this pseudo-variable (see 3.3) can have deep side effects on security of your applications.

```
window.ping // answers 'undefined'

var ping = function (string) {
  return string;
};

window.ping; // answers the ping function
window.ping('foo'); // answers 'foo'
```

Example 3.3: A function not attached to an object is attached to window

The example starts by checking that the window object has no property named ping. Then we define a new function at the top level, which is automatically added to the window object. Accessing window.ping returns the ping function. We check that we can effectively execute it.

3.3 this: an overly dynamic pseudo-variable

While at first sight, this may look like the pseudo-variable in Java, C# and Smalltalk (named self in this case), in JavaScript this has a special semantics. this is a dynamically bound builtin pseudo-variable of JavaScript. When used inside a function scope, the value of the pseudo-variable this is determined by the syntactic shape of the function invocation [MMT08, GSK10]. By syntactic shape, we mean that this is bound to a different object depending on the way the invocation expression is written.

In Example 3.4, while the same function is shared between two different objects, depending on the function invocation, this is bound to a different object.

```
var o = new Object();
// A new function is attached to 'o' on property 'f'
o.f = function() {return this;};

// f is invoked from o
o.f(); // answers o, so 'this' was bound to 'o'

var o2 = new Object();

// o2.f and o.f point to the same function object
o2.f = o.f;

// f is invoked from o2
o2.f(); // answers o2, so 'this' was bound to 'o2'
```

Example 3.4: Function invoking

The behavior described by the previous example looks natural and similar to the one of Java and C# where this is bound to the receiver. However, the semantics

behind this is more complex. Thus, depending on the way the function is invoked, this may not refer to the object which defines the function but to the global object window. As explained in section 3.2, every object not attached to another object is implicitly attached to the global object window.

The following example illustrates this behavior. First, a simple object, named o, with a method yourself is created that simply returns this. When the expression o.yourself is executed, o is returned as expected. Then the variable yourselfFunction is defined in the context of the global environment and its value is set to the method yourself. Now when yourselfFunction is executed, the global object window is returned instead of the object o, which defines the method.

We see that assigning a pointer to a function and invoking this function via this pointer changes what this is bound to inside the function.

```
// Creates a new object with a 'yourself' method
var o = {
  yourself: function() { return this; }
};

o.yourself() // answers o

//We attach o.yourself to window
var yourselfFunction = o.yourself;

yourselfFunction() // answers the 'window' object
```

Example 3.5: this and window interplay

As can be seen in Example 3.5 taken from [GSK10], one of the dangerous side effects of the this semantics is the ability to retrieve the window object from a function.

In this example, an object, named obj, is created with a property x and a method named setX: which mutates x. The return value of the expression window.x shows that window does not have a property x. Then the expression obj.setX(10) sets the value of the property x of obj to 10. Then the variable named f points to the mutator method of object obj.

Executing the mutator via the reference through f with 90 as a value will add a variable x to window and assign it 90. Here the syntax of the method invocation binds this to the global object window instead of obj. The value returned from the expression obj.x shows that the property x did not change and is still holding 10. Finally, the value returned from the expression window.x shows that the object window got a new property x holding the value 90.

```
var obj = {
  "x" : 0,
  "setX": function(val) { this.x = val }
};

window.x // answers 'undefined'
obj.setX(10);
obj.x // answers 10
var f = obj.setX;
f(90);
obj.x // answers 10 (obj.x was not updated)
window.x // answers 90 (window.x was created)
```

Example 3.6: this and window

This section has shown that this is dynamically bound and that its binding depends on the syntactic expression from which a function is invoked. In particular, imprecise use of this may lead to a security breach granting access to the top-level object window (and thus any object of the web browser frame).

3.4 Object constructors misuse

Constructors used without new. When invoked without the new keyword, the function gets evaluated in the global context. this is then bound to the window object as seen in Example 3.7.

```
var Person = function (name, surname, age) {
  this.name = name;
  this.surname = surname;
  this.age = age;
};

// Invoking the constructor as a simple function
var person = Person('John', 'Foo', 27);
person // answers 'undefined'
person.age // raises an error
window.surname // answers 'Foo'
```

Example 3.7: Not using the new keyword

Note that in Example 3.7, person is undefined since the constructor does not return the object. In addition window gets an additional surname property as shown by the last statement of the example.

Objects created using the same constructor will not share functions and data set in the constructor. Each object will have its own copy of the initialization values. Sharing can be achieved using the prototype as will be shown in section 4.1.

Constructors returning objects.

Example 3.8 shows that values returned by functions invoked as constructors (with the new operator) are ignored by the compiler.

```
var Dog = function () {
  this.name = 'milou';
  return 3; // this statement is ignored by the compiler
}

var dog = new Dog();
dog; // answers {name: 'milou'} as expected
```

Example 3.8: Returning a primitive type from a constructor function

Example 3.9 shows that when the returned object of a constructor function is not a primitive one, the constructor actually returns it. This behavior is unspecified by ECMAScript 5 and leads to misleading results. Constructor functions should never explicitly return anything. The new keyword takes care of returning the newly created object.

```
var Dog = function () {
  this.name = 'milou';
  return {name: 'tintin'}; // this statement is not ignored
}

var dog = new Dog();
dog; // unexpectedly answers {name: 'tintin'}
```

Example 3.9: Returning a non-primitive type from a constructor function

3.5 Lexical closures and unbound variables

JavaScript functions are lexical closures [MMT08, Dav06, GSK10]. Each lexical environment depends on its outer context. The closure scope grants access to the function arguments (accessed by value), as well as all variables accessible from the outer context scope, including all global variables.

In the Example 3.10, we show how a function has access to its outer scope's variables. outerFunction is a function which returns another function whose role is to increment a private variable localToOuterFunction and set the value to an object's property someProperty. We can see that innerFunction has access to the localToOuterFunction variable defined in outerFunction. We can also see that the two functions returned by the two calls to outerFunction have access to two different copies of localToOuterFunction.

```
function outerFunction(obj){
    var localToOuterFunction = 0;
    var innerFunction = function(){
        localToOuterFunction++;
        obj.someProperty = localToOuterFunction;
    }
    return innerFunction;
}
o = new Object();
returnedFunction = outerFunction(o);
returnedFunction();
returnedFunction();
o.someProperty // answers 2

o2 = new Object();
returnedFunction = outerFunction(o2);
returnedFunction();
o2.someProperty // answers 1
o.someProperty // answers 2
```

Example 3.10: A function and its outer scope

A naive use of closures may lead to issues as described in Example 3.11 where all handlers will unexpectedly always return the value 10.

```
var handlers = [];
for(var i=0; i < 10; i++) {
  handlers[i] = function() { return i; };
};
handlers[3](); // answers 10
```

Example 3.11: Variable scopes in closures #1

In the for loop we iterate over i from 0 to 10. In each iteration of the loop a closure is created and stored into the handlers array. When one of these closures is evaluated (the fourth one here), the value returned by the closure is the current value of i, not the value of i at the time of the closure creation.

The Example 3.12 illustrates how to use closures to solve the issue described before. This example is the same as the previous one, only surrounding the closure creation with an additional closure capturing the value of i inside the for loop. When one of the closures is evaluated, the expected number is returned.

```
var handlers = [];
for(var i=0; i < 10; i++) {
  (function (j) {
     handlers[j] = function() { return j; };
```

```
    })(i);
};
handlers[3](); // answers 3
```

Example 3.12: Variable scopes in closure #2

3.6 The with statement

As described in ECMAScript 3, the JavaScript with statement adds all properties of an object to the scope of the with statement as shown in Example 3.13.

```
var someGlobal = 'pilou';
var obj = new Object();
obj.propertyA = 1;

with (obj) {
    someGlobal = propertyA;
};

someGlobal; // answers 1
```

Example 3.13: Mixing scopes

The scope of the with statement includes all the variables of its outer scope (including global variables) and the object properties, overriding outer scope variables as shown in Example 3.14. In this example, inside the with statement, there is potentially two targets for the propertyA name: this name could be referring to either the global variable (with value 'property') or to the property of obj (with value 1). When using with, properties of the object passed as parameter to with always take precedence.

```
var propertyA = 'property';
var someGlobal = 'pilou';
var obj = new Object();
obj.propertyA = 1;

with (obj) {
    someGlobal = propertyA; // 'propertyA' is the property of obj
};

someGlobal; // answers 1
```

Example 3.14: Overriding outer scope variables

Using with is not recommended and is even forbidden in ECMAScript 5 strict mode. The recommended alternative is to assign the object's wanted properties to a temporary variable.

The dynamic nature of JavaScript combined with scope mixture lowers the predictability of JavaScript programs.

3.7 Lifted Variable Definitions

Local variables are automatically lifted to the top of the function in which they appear. For example in the following function foo(), the return statement has access to the variable x that was defined inside the if branch.

```
function foo() {
   if (true) {
      var x = 10;
   }
   return x;
}

foo(); // answers 10
```

This happens because Javascript automatically rewrite the previous code to something like the following:

```
function foo() {
   var x;
   if (true) {
      x = 10;
   }
   return x;
}

foo(); // answers 10
```

Such behavior can lead to unintuitive results as demonstrated in the following example.

```
function foo(x) {
   return function() {
      var x = x;
      return x;
   }
}

foo(200)(); // answers undefined
```

The function returned by the function foo does not return the value passed to foo but undefined (the value of unaffected variables). In this last example we might expect the x on the right-hand side of var x = x to reference the argument x of foo.

Nevertheless, due to lifting, all bound occurrences of x in the nested function refer to the local variable x as is made clear in the following rewrite:

```
function foo(x) {
   return function() {
      var x; // JavaScript splits "var x = x" in two statements
      x = x;
      return x;
   }
}

foo(200)(); // answers undefined
```

The expression var x = x reads and writes back the initial value of x (*i.e.,* undefined). var x = x works as var x; x = x; hence the right-hand x of the assignment refer to the local variable x.

3.8 Type coercion

JavaScript performs type coercion (implicit type conversion) over objects on, for example, comparisons and if statements checks. The automatic type coercions performed by JavaScript are well known causes of bugs which lower the robustness of JavaScript applications.

The basic rules about type coercion are [ECM11]:

Boolean coercion. When expecting a boolean value, as in the if statements, JavaScript transforms the result of our expression automatically into a boolean value. Values equivalent to false are null, undefined, false, +0, -0, NaN and the empty string. The rest of the JavaScript objects are coerced to true.

```
var falsyValue = false;
if(!"") {
  falsyValue = true;
}
falsyValue // Answers true

falsyValue = false;
if(0) {
  falsyValue = true;
}
falsyValue // Answers false
```

Equality coercion. When two objects are compared (as via the equality operator ==), depending on their types, none, one or both objects are coerced before being compared. After coercion, if both operands have the same type, the comparison is

finally resolved by the strict equality operator. Depending on the type, valueOf() or toString() may be implicitely performed on the compared objects. In addition, when performing an equality comparison, the following objects are considered as false: null, 0, false, ” and NaN.

```
false == 0 // answers true

0 == false // answers true

"" == 0 // answers true

false == "" // answers true

{} == {} // answers false

var n = {
   valueOf: function () {
      return 1;
   },
   toString: function () {
      return "2";
   }
};

n == 1; // answers true
n == "2"; // answers false

var n = {
   toString: function () {
      return "2";
   }
};

n == 1; // answers false
n == "2"; // answers true

[ [ [ 42 ] ] ] == 42; // answers true. valueOf() an array with one
    element answers its element

true + 3; // answers 4
```

Example 3.15: Some unintuitive examples of type coercion

Strict equality coercion. The strict equality operator === compares both type and value of the operands, without performing type coercions. It only resolves to true, when both operands have the same type and value. The only exception are non-primitive objects, which are strictly equal if they are the same object.

```
false === 0 // answers false
0 === false // answers false
"" === 0 // answers false
false === "" // answers false

1 === 1 // answers true
{} === {} // answers false
```

Chapter4. JavaScript programming practices

This section presents different coding practices in JavaScript development (ECMAScript 3) that result in robust, extensible and understandable sofware. In particular, we stress the semantics of the new constructor and its impact on inheritance [Dou08].

4.1 Defining prototypes

JavaScript as defined in ECMAScript 3 is a prototype-based object-oriented language where each object has a prototype (referenced in the __proto__ core property). The object inherits all properties from it. Since the prototype is also an object, the prototype chain inheritance is similar to a class hierarchy in class-based object-oriented languages.

Constructors structure object creation and initialize properties. Each time an object is created, a different copy of each attribute specified in the constructor is assigned to the object. When sharing is needed between objects, the shared properties must be defined in the constructor's prototype as can be seen in Example 4.1 and Figure 4.1.

```
var Cat = function (color, name) {
  this.color = color;
  this.name = name || 'default name';
}

Cat.prototype.numberOfLegs = 4;

var garfield = new Cat('red', 'Garfield');
var azrael = new Cat('black', 'Azrael');

garfield.color; // answers 'red'
garfield.numberOfLegs; // answers 4

azrael.color; // answers 'black'
azrael.numberOfLegs; // answers 4

Cat.prototype.numberOfLegs = 5;
garfield.numberOfLegs; // answers 5
azrael.numberOfLegs; // answers 5

azrael.color = 'grey';
garfield.color; // answers 'red'
```

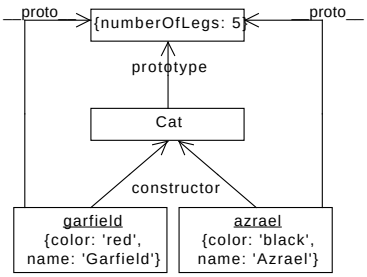

Figure 4.1: Sharing a property between objects (see Example 4.1)

```
azrael.color; // answers 'grey'
```

Example 4.1: Sharing a value between objects through their prototype (see Figure 4.1)

When a new object is created by calling a constructor, its __proto__ core property is initialized to the constructor's prototype property value. In Example 4.2 we set the prototype of function Dog to be an object created from function Animal (see Figure 4.2). Note that we have to set the constructor of the prototype to be the function Dog. Then when an object created from the function Dog is asked for a property that it does not define locally, the lookup is performed following the prototype chain (*i.e.*,, looking inside the __proto__ core property value). Here the property isAnAnimal is found in the prototype of Dog which is an object created from the Animal constructor.

```
var Animal = function () { };

Animal.prototype.isAnAnimal = true;

var animal = new Animal();

var Dog = function () {};

// The prototype of Dog is set to a new Animal,
// so that future Dog objects will inherit from Animal.prototype
Dog.prototype = new Animal();

// We need to manually change Dog.prototype.constructor so that
// future Dog objects will have Dog as constructor
```

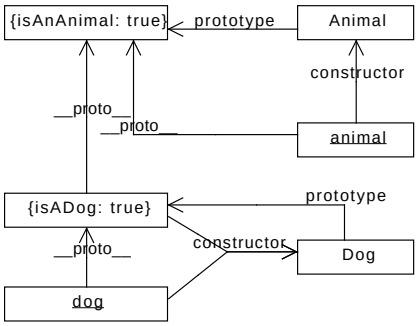

Figure 4.2: Prototypical inheritance (see Example 4.2)

```
// (instead of Animal).
Dog.prototype.constructor = Dog;

// All Dog objects must share this property
Dog.prototype.isADog = true;

var dog = new Dog();
dog.isAnAnimal; // answers true
dog.isADog; // answers true
```

Example 4.2: Inheritance in JavaScript prototypical programming (see Figure 4.2)

Accessing overridden functions. Other object-oriented languages have a message resend mechanism, often implemented as *super* sends. To perform *super* sends in JavaScript, the lookup must be explicitly forwarded to the prototype (see Example 4.3 and Figure 4.3).

```
// 'Object' being a function, we add a new method to all objects
Object.prototype.superSend = function (selector, args) {
  // We use 'inheritsFrom' to reference the prototype and we search
  // the property in variable 'selector' from this prototype:
  return this.inheritsFrom[selector].apply(this, args);
};

var Animal = function () { };
Animal.prototype.say = function (string) {
   return 'hello ' + string;
};
```

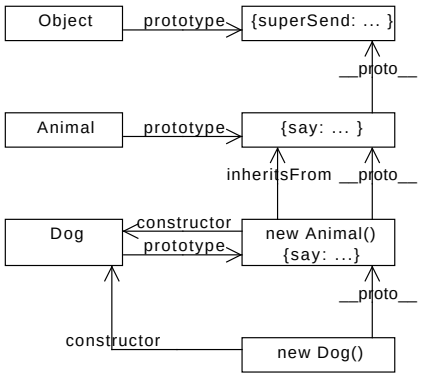

Figure 4.3: Message resending – super sends (see Example 4.3)

```
var Dog = function () { };
Dog.prototype = new Animal();
// We add our own property to retain inheritance
// without using the not standard __proto__
Dog.prototype.inheritsFrom = Dog.prototype.constructor.prototype;
Dog.prototype.constructor = Dog;

new Dog().inheritsFrom === new Dog().__proto__.__proto__; // answers true

Dog.prototype.say = function (string) {
   return this.superSend('say',['wouf wouf']);
};

new Dog().say("I'm a dog"); // answers 'hello wouf wouf'
```

Example 4.3: Message resending – super sends (see Figure 4.3)

The dynamic capabilities of JavaScript allow the usage of this mechanism to extend existing objects like *Arrays*, *Numbers*, and *Functions*, through the addition of properties and methods to their prototype constructors.

4.2 Closures and 'functional' inheritance

We've previously shown an example of how to provide an inheritance-like relation in JavaScript using prototypes, allowing us to share properties and methods between our objects. Unfortunately, the builtin inheritance mechanism of ECMAScript 3 has several drawbacks: (1) it depends on many implementation details that could lead to several mistakes, (2) it does not provide any access protection

between objects, their prototypes, and the outer scope.

In Example 4.4 a JavaScript idiom appears showing how we can use closures to support visibility principles such as private attributes and inheritance [Dou08] instead of the typical prototype relationship we just described. The main idea is to create functions that create and initialize an object – declared as that in the example – with default values and methods. The values set in the object's properties will be public values. We can also define and use variables and functions with the var keyword to make them private to our object. To customize the initialization, we can also pass an object as a parameter to this function as a spec hash-table.

```
var animal = function (spec) {
  // We take either the parameter or the empty object if
  // spec is null
  spec = spec || {};
  var that = {};

  // Initialization
  that.isAnAnimal = true;

  // Private
  var name = spec.name || 'unnamed';

  // Public
  that.getName = function() {
    return name;
  };

  return that;
};

var dog = function (spec) {
  spec = spec || {};
  var that = animal(spec); // makes dog inherits from animal

  that.isADog = true;

  return that;
};

var aDog = dog({name: 'milou'});
aDog.isAnAnimal; // answers 'true'
aDog.isADog; // answers 'true'
aDog.getName(); // answers 'milou'
aDog.name; // answers 'undefined'
```

Example 4.4: Using closures to support access visibility to properties

Chapter5. ECMAScript 5

We also present an overview of the features of ECMAScript 5 [ECM97, ECM11].

Released in June 2011, ECMAScript 5 defines the latest standarized version of the JavaScript language. This release includes improvements and clarifications compared to ECMAScript 4. In this section we propose a survey of two important aspects of ECMAScript 5: object creation and properties access. These aspects improve object encapsulation, giving a finer-grained object-property definition and thus improving security.

5.1 Object creation

ECMAScript 5 offers a new way to instantiate objects with the ability to specify the new object's prototype and properties.

```
var objOld = new Object();
var objNew = Object.create(null); // new function from ECMAScript 5

// a.isPrototypeOf(b) checks if 'a' is in the __proto__
    inheritance
// chain of b (i.e., b is derived from a)
Object.prototype.isPrototypeOf(objOld); // answers true
Object.prototype.isPrototypeOf(objNew); // answers false

objOld.toString; // answers a function
objNew.toString; // answers 'undefined'
```

Example 5.1: Creating an object with null as prototype

Example 5.1 shows how to create a new object named objNew that has no prototype and no inherited property.

When passed an argument, Object.create sets the __proto__ property of the object to the argument. As a result, new Object() is equivalent to Object.create(Object.prototype).

Object.create also optionally accept a second argument that is an object whose properties serve to initialize the new object. In Example 5.2, the object obj inherits all standard properties from Object.prototype and defines a new property foo whose value is the string "hello".

```
var obj = Object.create(Object.prototype, {
  foo: { writable: true, configurable: true, value: "hello" },
});
obj.__proto__ === Object.prototype; // answers true
obj.toString; // answers a function
```

```
obj.foo; // answers "hello"
```

Example 5.2: Creating an object with prototype and a property

The keys writable and configurable are described below.

5.2 Defining object properties

Security wise, ECMAScript 3 doesn't have any concept of private properties.[1] All object properties are publicly visible, inherited and modifiable at will. ECMAScript 5 improves the situation by introducing a fine-grain protocol to define object properties.

Object.defineProperty is one of the core changes to JavaScript defined in ECMAScript 5. This function takes three parameters: the object on which the property is defined, a string (name of the new property) and a descriptor object. The descriptor object can be a data descriptor or a getter/setter descriptor. A descriptor object can have the following optional keys:

- enumerable: if true, the property shows up during enumeration of the properties of the receiver;
- configurable: if true, the property can be deleted, and the descriptor can be changed afterwards.

In case of an accessor descriptor, two keys get and set can be used to define accessor methods to the property.

In case of a data descriptor, two keys value and writable can be used to respectively set an initial value and specify if the property can be written.

```
var dog = {};
Object.defineProperty(dog, 'name', {
    enumerable: true,
    configurable: false,
    value: 'Pilou',
    writable: false
});

dog.name; // answers 'Pilou', the default value
dog.name = 'another name'; // tries to set a new value
dog.name; // answers 'Pilou' as the property is not writable

delete dog.name; // tries to remove the property from the object
```

[1] We call private properties object properties that are neither enumerable nor writable.

```
dog.name; // answers 'Pilou' as the property is not configurable
```

Example 5.3: Defining properties

Example 5.3 shows how to use Object.defineProperty. First an empty object dog is created. A property 'name' is added to dog and set a default value of 'Pilou'. This property is neither configurable nor writable. As a result, trying to change the value of the property dog.name or trying to delete it both fail.

Object.preventExtensions. ECMAScript 5 introduces two important functions regarding object extensions: preventExtensions and isExtensible, both available from Object. As seen in Example 5.4 preventExtensions takes an object as argument and prevents any further addition of properties. Existing properties may still be deleted though. isExtensible is a testing function answering a boolean value that indicates whether properties can be added or not.

```
var dog = {};
Object.defineProperty(dog, 'name', {
    enumerable: true,
    configurable: false,
    value: 'Pilou',
    writable: false
});

Object.isExtensible(dog); // answers true
Object.preventExtensions(dog);
Object.isExtensible(dog); // answers false

dog.age = 5; // tries to add a new property to 'dog'
dog.age // anwers undefined because 'dog' is not extensible
```

Example 5.4: Preventing object extensions

ECMAScript 5 also introduces full immutability of objects through Object.freeze, and can be tested with Object.isFrozen, and seen in Example 5.5.

```
var dog = {};
Object.defineProperty(dog, 'name', {
    enumerable: true,
    configurable: false,
    value: 'Pilou',
    writable: false
});

Object.isFrozen(dog); // answers false
Object.freeze(dog);
```

```
Object.isFrozen(dog); // answers true

dog.age = 5; // tries to add a new property to 'dog'
dog.age // anwers undefined because 'dog' is not extensible

delete dog.name // answers false
dog.name // answers 'Pilou'
```

Example 5.5: Object immutability

By adding the functions mentioned in this section (notably create, defineProperty, preventExtensions and freeze), ECMAScript 5 makes it possible for developers to secure their objects.

Chapter6. Key Bibliography Elements

Here we present the key and limited articles that we encourage you to read if you want to get deeper into the semantics of JavaScript.

- Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS '08, pages 307-325, Berlin, Heidelberg, 2008. Springer-Verlag. The paper presents a 30-page operational semantics, based directly on the JavaScript specification. The semantics covers most of JavaScript directly, but does omit a few syntactic forms. They discuss various differences between the standard and implementations.
- Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP'10, pages 126-150, Berlin, Heidelberg, 2010. Springer-Verlag. In this article a core calculus is defined. Based on it, several aspects of JavaScript are described. Some badly designed features of Javascript are described.
- ECMAScript version 3.0 specification (188 pages). http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf
- ECMAScript version 5.1 specification (255 pages). http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf

Chapter7. Conclusion

This deliverable has introduced an overall picture of JavaScript as defined in ECMAScript 3 and ECMAScript 5, focusing on security aspects of its semantics. We offered a detailed review of JavaScript subtle points concerning security and application predictability.

We have seen that the window object is key to data privacy protection. Nevertheless, we have pointed out how JavaScript exposes window to the entire environment constraining any security approach.

We have described scoping issues and the possible resulting security leaks. For example, the this pseudo variable is bound to a different object depending on the way the function invocation expression is written (syntactic shape of the function invocation in which this is used). This dynamic behavior can be exploited by an attacker to leak private objects. As another example, the fact that JavaScript lifts variable definitions inside functions (*i.e.,* moves to the top) leads to unsuspected variable shadowing, dramatically lowering behavior predictability.

In the last chapter, we have proposed a description of ECMAScript 5 features regarding object declaration and property access that improve encapsulation, therefore securing exposure of objects.

In the following deliverable, we will leverage this knowledge, detailing existing sandboxing techniques, and for each of them, we will review its advantages and weaknesses in different contexts.

Bibliography

[Dav06] Flanagan David. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., fifth edition, 2006.

[Dou08] Crockford Douglas. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[ECM97] ECMA, editor. *ECMAScript Language Specification*. European Computer Machinery Association, June 1997.

[ECM11] ECMA, editor. *ECMAScript Language Specification version 5*. European Computer Machinery Association, July 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[MFF01] Lynette I. Millett, Batya Friedman, and Edward Felten. Cookies and web browser design: toward realizing informed consent online. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '01, pages 46–52, New York, NY, USA, 2001. ACM.

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[Mog02] Jeffery C. Mogul. Clarifying the fundamentals of http. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 25–36, New York, NY, USA, 2002. ACM.

[MvD07] Ali Mesbah and Arie van Deursen. Migrating multi-page web applications to single-page ajax interfaces. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 181–190, Washington, DC, USA, 2007. IEEE Computer Society.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of Ecoop 2011*, 2011.